Available online at www.sciencedirect.com

SciVerse ScienceDirect

Procedia Computer Science 18 (2013) 250 – 259

Procedia Computer Science

International Conference on Computational Science, ICCS 2013

An Empirical Evaluation of the Cost and Effectiveness of Structural Testing Criteria for Concurrent Programs

Maria A. S. Brito[a], Simone R. S. Souza[a], Paulo S. L. Souza[a,*]

[a]*Universidade de São Paulo, Instituto de Ciências Matemáticas e de Computação, ICMC/USP, P.O. 668, São Carlos (SP), Brazil, 13560-970*

Abstract

Concurrent program testing is not a trivial task. Features like nondeterminism, communication and synchronization impose new challenges that must be considered during the testing activity. Some initiatives have proposed testing approaches for concurrent programs, in which different paradigms and programming languages are considered. However, in general, these contributions do not present a well-formed experimental study to validate their ideas. The problem is that the data used and generated during the validation is not always available, hampering the replication of studies in the context of other testing approaches. This paper presents an experimental study, taking into account the concepts of the Experimental Software Engineering to evaluate the cost, effectiveness and strength of the structural testing criteria for message-passing programs. The evaluation was conducted considering a benchmark composed of eight MPI programs. A set of eight structural testing criteria defined for message-passing programs was evaluated with the ValiMPI testing tool, which provides the support required to apply the investigated testing criteria. The results indicate the complementary aspect of the criteria and the information about cost and effectiveness has contributed to the establishment of an incremental testing strategy to apply the criteria. All material generated during the experimental study is available for further comparisons.

Keywords: Software testing; Concurrent programs; MPI programs; Experimental study

1. Introduction

The demand for distributed and parallel applications has been growing due to the advanced hardware technology, which provides more efficient machines and allows to process large volumes of data. However, these applications are inevitably more complex than sequential ones. Every concurrent software contains features, such as nondeterminism, synchronization and inter-process communication, which significantly hamper their validation and their testing. Approaches to test concurrent programs efficiently and effectively have been proposed and are an important factor for the success and quality of the programs built in this domain.

Several studies have been conducted to define approaches to test concurrent programs [1, 2, 3, 4, 5]. In general, these studies present some evaluation to demonstrate the applicability of their contribution. The problem is that the data used is not always available, hampering the replication of the studies and the fair comparison among different testing techniques for concurrent programs. In the context of sequential program testing, Weyuker [6] pointed out the importance of comparative studies to evaluate different testing techniques, allowing the replication

*Corresponding author. Tel.: +55-16-3373-9700 ; fax: +55-16-3373-9700
E-mail address: masbrit@icmc.usp.br; srocio@icmc.usp.br; pssouza@icmc.usp.br.

Selection and peer review under responsibility of the organizers of the 2013 International Conference on Computational Science
doi:10.1016/j.procs.2013.05.188

of the experimental studies. Thus, demonstrating the applicability and effectiveness of testing techniques is as important as proposing testing techniques for new application domain.

The empirical evaluation of techniques, criteria and testing tools has been intensified in recent years, mainly in the context of traditional programs. These empirical evaluations, in general, consider three basic factors for a comparison: cost, effectiveness and strength [7]. Cost refers to the effort required to satisfy a testing criterion and can be measured by the number of test cases necessary to cover it. Effectiveness refers to the ability of a test set to reveal defects. Strength refers to the probability to satisfy a testing criterion using a test set adequate to another testing criterion. These comparison factors are important for the proposition of an efficient testing strategy taking into account the benefits of each testing criterion.

This paper has contributed in this direction, presenting an experimental study that evaluates structural testing criteria defined for message-passing programs, analyzing their cost, effectiveness and strength. Experimental study takes into account the process defined by Wholin [8], which includes activities for the definition, planning, conduction, analysis and packing of experimental studies. A benchmark composed of eight MPI (Message Passing Interface) programs is defined and used in our study. MPI is a message-passing library interface specification for the development of portable message-passing concurrent programs using sequential languages, such as C and Fortran [9]. Our work considers MPI programs written in C language.

The analyzed testing criteria were proposed by Souza et al. [10] and include structural criteria to test concurrent and sequential aspects of message-passing programs. Information about control, data and communication flows is extracted from a program under test and used to guide the generation of a test case set. Eight different testing criteria were analyzed using the ValiMPI support tool [11]. This tool provides the required resources to apply test cases and evaluate their coverage in programs considering the structural testing criteria for MPI programs.

The material generated during the experimental study, including programs, results of testing activity, ValiMPI tool and results of experimental study has been organized and is available for public access, providing relevant information to further studies and comparisons. According to our knowledge, it is the first study which uses concepts of the Experimental Software Engineering for the definition, conduction and analysis of the testing criteria in the context of concurrent programs.

The remaining of the paper is organized as follows: Section 2 presents the test model and the structural testing criteria for message-passing programs, as well as the ValiMPI testing tool. Section 3 describes the experimental study, including the definition, planning, results and analysis. Section 4 reports the related works and Section 5 presents the final considerations and future research directions.

2. Structural Testing Criteria for Message-Passing Programs

In message-passing programs, communication is made by *send* and *receive* basic primitives, in which a process can send a message to another process or to a group of processes. The first one is called point-to-point communication and the second is called collective communication. In both cases, the test activity must establish an association between the variables sent and the location where these variables are used in the receiver process(es). Besides, it is important to define a strategy to derive all possible synchronizations among the processes of the concurrent software processes. Executing these distinct synchronizations allows the verification of different pairs of definition and use of variables in different processes. Associated to these synchronizations there are important questions that must be considered, such as non-determinism, controlled execution and race conditions.

Souza et al. [10] defined a test model and a set of structural testing criteria that represent the main features of message-passing programs, including control, data and communication aspects. The test model considers that a concurrent program is a set $Prog = p^0, p^1, ..., p^{n-1}$ composed of its n parallel processes. A Control Flow Graph (CFG) of each process p is generated using the same concepts of sequential programs. Each CFG^p represents the control flow of a process p. A Parallel Control Flow Graph (PCFG) generated for $Prog$ is composed of CFG^p (to $p = 0 \ldots n-1$) and edges of communication among processes. N represents the set of nodes and E represents the set of edges in PCFG. E has two subsets: E_p, which are edges of a same process and E_s, composed of edges that represent the communication between processes, called interprocess edges. A node i in a process p is represented by notation n_i^p. Two subsets of N are defined: N_s, which are the nodes composed of primitives *send* and N_r, which are nodes composed of primitives *receive*. A set R_i^p is associated with each $n_i^p \in N_s$ containing possible nodes that can receive the message sent by node n_i^p. This set is important to establish all possible communication pairs.

A path π in CFG^p is called intraprocess if it contains no interprocess edges and is composed of a sequence of nodes $\pi = (n_1, n_2, ..., n_m)$ in which $(n_i, n_{i+1} \in E_i^p)$. A path π that contains at least one interprocess edge is called an interprocess path.

In relation to data flow information, the concepts employed in sequential programs are considered and extended to include the communication between processes. Thus, in addition to the computational and predicative use of variables (c-use and p-use), the communication use (s-use) is defined and associated with the variables in a sent message. Considering these concepts, the following associations between definition and use of variables are defined:

a) *s-use association*: defined by a triple $(n^{p1}, (m^{p1}, k^{p2}), x)$, such that $x \in def(n^{p1})$ and (m^{p1}, k^{p2}) has an s-use of x and there is a definition-clear path with respect to x from n^{p1} to (m^{p1}, k^{p2}).

b) *s-c-use association*: given by $(n^{p1}, (m^{p1}, k^{p2}), l^{p2}, x^{p1}, x^{p2})$, where there is an s-use association $(n^{p1}, (m^{p1}, k^{p2}), x^{p1})$ and a c-use association (k^{p2}, l^{p2}, x^{p2}).

c) *s-p-use association*: given by $(n^{p1}, (m^{p1}, k^{p2}), (n^{p2}, m^{p2}), x^{p1}, x^{p2})$, in which there is an s-use association $(n^{p1}, (m^{p1}, k^{p2}), x^{p1})$ and a p-use association $(k^{p2}, (n^{p2}, m^{p2}), x^{p2})$.

Based on the information derived from the test model, a family of structural testing criteria for message-passing programs was defined in details in Souza et al. [10]. The following testing criteria are considered in our experimental study:

1. **all-nodes-s criterion (ans)**: the test sets must execute paths that cover all nodes $n_i^p \in N_s$.
2. **all-nodes-r criterion (anr)**: the test sets must execute paths that cover all nodes $n_i^p \in N_r$.
3. **all-nodes criterion (an)**: the test sets must execute paths that cover all nodes $n_i^p \in N$.
4. **all-edges-s criterion (aes)**: the test sets must execute paths that cover all edges $(n_j^{p1}, n_k^{p2}) \in E_s$.
5. **all-edges criterion (ae)**: the test sets must execute paths that cover all edges $(n_j, n_k) \in E$.
6. **all-c-uses criterion (acu)**: the test set must execute paths that cover all c-use associations.
7. **all-p-uses criterion (apu)**: the test set must execute paths that cover all p-use associations.
8. **all-s-uses criterion (asu)**: the test set must execute paths that cover all s-use associations.

The ValiMPI tool supports the application of these testing criteria, providing functionalities to generate a test session, save and run test data and evaluate the testing coverage for a given criterion [11]. The ValiMPI extracts control, data and synchronization information from a source code of the program to generate the required elements of each testing criterion. The program is instrumented by inserting *check-points* that allow to generate an execution trace, in which it is possible to evaluate the coverage obtained by test cases in relation to testing criteria. Functionalities are available to execute a test case with all the possible synchronization sequences, assuring their coverage during the test activity.

3. Experimental Study

Considering the objective of this experimental study, the following goals were defined based on guidelines for Experimental Software Engineering proposed by Wohlin [8]:

- **Object of study**: the testing criteria for message-passing programs all-nodes, all-nodes-r, all-nodes-s, all-edges, all-edges-s, all-c-uses, all-p-uses and all-s-uses;
- **Purpose**: evaluation of the application cost, effectiveness and strength of each testing criterion;
- **Quality focus**: cost, effectiveness and strength of each testing criterion;
- **Perspective**: viewpoint of the researcher;
- **Context**: this experimental study was carried out by us considering a set of programs of calculus, physics and bioinformatics, using ValiMPI testing tool to support the application of the analyzed testing criteria.

3.1. Planning

We selected a benchmark composed of eight representative programs, seven of them are classical problems in concurrent programming and one of them is a real concurrent program from bioinformatics domain, proposed by Bonetti et al. in [12]. The eight programs are:

1. **GCD**: this program calculates the greatest common divisor considering three numbers, according to algorithm presented in [13].
2. **Jacobi**: this program implements the iterative method of Gauss–Jacobi for solving a linear system of equations.
3. **Mmult**: this program implements the multiplication of a matrix using domain decomposition, according to algorithm described in [14].
4. **Qsort**: this program implements the recursive quicksort method, according to algorithm presented in [15].
5. **Reduction**: this program implements the reduction operation of distributed data.
6. **Sieve**: this program implements the sieve of eratosthenes, according to algorithm described in [14].
7. **Trap**: this program calculates an approximation of the integral of a function x via trapezoidal method.
8. **Van der Waals**: real application developed by Bonetti et al. in [12], which calculates the Van der Waals energy of a protein using concepts of genetic algorithms.

These programs were implemented in C/MPI and, in order to standardize the size of these programs, each one is composed of four parallel processes. Information about the complexity of each program is presented in Table 1:

Table 1. Characteristics of the case studies.

Program	LOC	Send Primitives	Receive Primitives
GCD	112	5	5
Jacobi	691	11	19
Mmult	192	9	15
Qsort	480	7	13
Reduction	132	1	1
Sieve	113	3	7
Trap	77	1	1
Waals	540	4	4

3.1.1. Hypotheses

Considering the objectives of this study, we defined the research question and a set of hypotheses, which were used to analyze the results.

Research question*: what are the effectiveness, application cost and strength of the testing criteria for message-passing programs?*

Null Hypothesis 1 (NH1): the application cost is the same for all structural testing criteria analyzed.

Alternative Hypothesis 1 (AH1): the application cost is different for at least one structural testing criterion for message-passing programs.

Null Hypothesis 2 (NH2): the effectiveness is the same for all structural testing criteria analyzed.

Alternative Hypothesis 2 (AH2): the effectiveness is different for at least one structural testing criterion for message-passing programs.

Null Hypothesis 3 (NH3): in relation to strength, no testing criterion subsumes another.

Alternative Hypothesis 3 (AH3): there is at least one testing criterion that subsumes another.

3.1.2. Variables Selection

In this section we present the dependent and independent variables that are used to represent the treatments of the experiment. The independent variables are the input of the experiment and are manipulated and controlled. The dependent variables are the response of the independent variables and represent the effect of the changes in the independent variables.

In this study, the independent variables of interest are: a) structural testing criteria for concurrent programs, b) programs used; and c) faulty programs.

In relation to dependent variables, the following variables are of interest: a) number of required elements of each testing criterion; b) size of the test set adequate to each testing criterion; c) number of infeasible elements of each testing criterion; d) number of defects revealed by each testing criterion; and e) percentage of coverage obtained to satisfy each testing criterion.

3.2. Preparation of the Experiment

Some tasks were carried out before the execution of the experiment, including: preparation of the programs, definition of set of defects to be considered, definition of how the defects would be inserted in the programs and installation of the ValiMPI testing tool.

The environment for the development of the experimental study has the following features: GNU/Linux operational system using the distribution Ubuntu 10.04 with the kernel 2.6.32, compiler *gcc* 4.1.2, Open MPI 1.4 and ValiMPI testing tool.

3.3. Execution of the Experiment

Initially, adequate test sets were manually generated for the programs, where each test set traverses all feasible required elements of a particular testing criterion. The programs were executed with the test sets using the ValiMPI tool and observing the coverage obtained. New test cases were added until all feasible required elements had been executed. In this phase, the infeasible elements were manually identified. The cost was evaluated considering the size of the adequate test sets. Also, information about the quantity of infeasible required elements were used to calculate the cost of each testing criterion.

To evaluate the effectiveness, defects were manually inserted into the programs, based on classifications of defects proposed by DeSouza et al. [16] and Agrawal et al. [17]. These classifications are based on errors made by developers, being relevant for our experimental study. Following a strategy similar to mutation testing, the defects were systematically inserted in each program, generating one program version for each inserted defect, totaling 334 faulty programs. Table 2 shows the number of defects inserted in each program, according to the type of defect considered. Some kind of defects did not generate faulty versions for some programs. Due to the features of each program, some defects did not make sense to be applied or the defects have generated an erroneous version, for instance, a deadlock in the program. In this case, the faulty versions were not considered.

These faulty programs were executed with the adequate test set and the ability to reveal the defects was registered.

The adequate test sets were also used to evaluate the strength of the criteria, in which a test set T_{c1}, adequate to criterion $C1$, is evaluated in relation to criterion $C2$; if T_{c1} is adequate to $C2$, then $C1$ may include $C2$. Thus, if the coverage of T_{c1} to $C2$ is low, this means that the criterion $C1$ has higher strength otherwise, $C1$ has lower strength in relation to $C2$. In this phase only the original (correct) programs were considered.

Table 2. Quantity of defects inserted in each program.

Type of defect	GCD	Jacobi	Mmult	Qsort	Reduction	Sieve	Trap	Waals
incorrect loop or selection structure	2	8	0	2	3	1	2	1
incorrect process in messages	2	4	1	0	0	2	1	0
source process changed by ”any” process in messages	1	5	0	2	0	1	0	0
incorrect size of array	2	4	3	1	0	4	0	0
non initialized variable	0	5	4	0	3	1	4	2
incorrect data types	0	0	0	0	0	0	0	3
incorrect size of message	0	2	4	1	0	2	4	2
incorrect message address	2	4	2	4	4	6	3	4
incorrect type of parameter	1	3	3	1	1	1	1	0
incorrect message data type	1	1	1	1	0	1	1	2
replacement blocking by non-blocking message	5	1	1	2	2	4	1	2
change of operator in the variable definition	2	8	5	3	2	8	7	5
incorrect data sent or received	1	5	5	3	2	5	3	4
change of the logical operator in predicative statements	2	5	3	5	1	7	2	4
missing statements	5	8	6	4	2	7	5	5
incorrect variable definition	4	4	4	5	1	7	0	5
increment/decrement of variables in messages	0	4	2	1	0	3	0	0
Total	**30**	**71**	**44**	**35**	**22**	**60**	**33**	**39**

3.4. Analysis of Results

This section presents some analysis related to the hypotheses formulated, where the results of statistical tests admitted a 95% significance level. For space reasons, some results are not presented here; more details and all the material used in the experiment can be obtained in: www.labes.icmc.usp.br/~experiments/MPIConcurrentCriteria.rar.

3.4.1. Data Analysis - Cost

Hypotheses $NH1$ and $AH1$, related to the application cost, were evaluated considering number of required elements, number of infeasible elements and size of the adequate test set.

Figure 1 shows the boxplot of the cost, considering the size of adequate test sets. A boxplot graphically represents five aspects of a data distribution (from the bottom to the top): the smallest observation, lower quartile, median, upper quartile, and largest observation. It can also indicate outliers for some data of the distribution (see points in boxplot in Figure 2). We can observe that the medians are higher for the testing criteria *all-s-uses (asu)*, *all-p-uses (apu)* and *all-edges (ae)*, indicating that the number of test cases necessary to satisfy these criteria is larger than for the other criteria.

The ShapiroWilk indicated that the population is distributed normally, thus the ANOVA methods was employed to this statistical analysis. We have obtained a *p-value* < 0.05 and the hypothesis $AH1$ is accepted indicating that at least one testing criterion presents a different value to the cost against the other criteria. Thus, we have applied the Tukey test for multiple comparisons to verify which criteria have significant difference in cost. The results indicate a significant difference between the following pairs of criteria: 1) *all-s-uses* and *all-nodes-r* and 2) *all-s-uses* and *all-nodes-s*. This result is according to the boxplot in Figure 1 demonstrating that the *all-s-uses* is the costlier criterion and the *all-nodes-r* and *all-nodes-s* are the lower cost criteria.

Considering required elements and infeasible elements, the results of the cost evaluation are similar and, therefore hypothesis $AH1$ can be accepted.

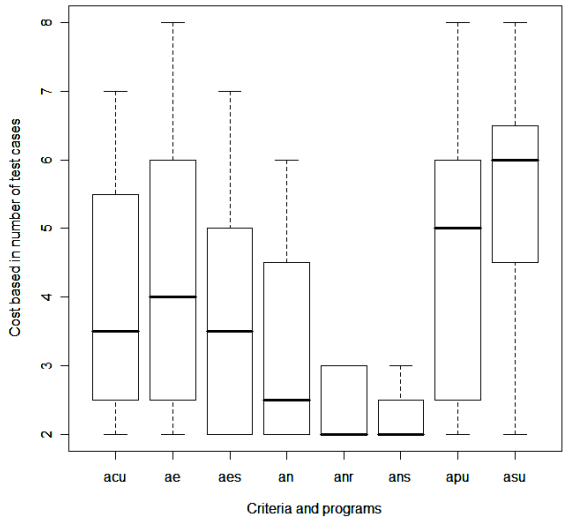

Fig. 1. Boxplot of the cost based on the size of adequate test. sets

3.4.2. Data Analysis - Effectiveness

In relation to effectiveness, the hypotheses $NH2$ and $AH2$ were evaluated (Figure 2). The boxplot in Figure 2 shows evidences that some criteria are more effective than others. For instance, the *all-s-uses (asu)* criterion is able to reveal 100% of defects injected in the programs, except for one program, whose effectiveness was 95.8% (outlier). They also show that the medians are higher for the testing criteria *all-p-uses (apu)*, *all-c-uses (acu)* and *all-edges (ae)*.

An important result is that the *all-edges* is an effective criterion to reveal faults in context of concurrent programs, contradicting the results obtained when this criterion is considered for sequential programs. This result became interesting because the cost of this criterion is lower, motivating further investigation of this testing criterion.

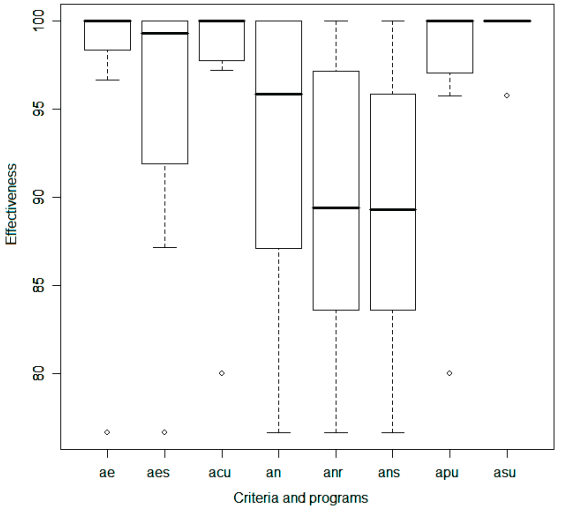

Fig. 2. Boxplot for the effectiveness of the testing criteria.

3.4.3. Data Analysis - Strength

In relation to strength, the hypotheses $NH3$ and $AH3$ were evaluated using the statistical method of cluster analysis. The hierarchical cluster analysis splits a sample in groups based on similarities between them. A dendrogram is a technique of the cluster analysis and it groups the data using hierarchical trees and measurements as the average, for example. In this sense, we have applied this analysis for each testing criterion and we have generated a dendrogram for each testing criterion studied.

Figure 3 presents the dendrogram chart for *all-s-uses (asu)* criterion, in which the adequate test set to this criterion was applied to all the other criteria. As result, all testing criteria in the same level of *all-s-uses* criterion obtained 100% coverage by the adequate test set to this criterion. This happens for the criteria *all-nodes (an)*, *all-nodes-s (ans)* and *all-nodes-r (anr)*. Thus, in this experimental study the *all-s-uses (asu)* criterion includes the criteria *all-nodes-s (ans)*, *all-nodes (an)* and *all-nodes-r (anr)*.

One dendrogram chart as been made for each testing criteria and the following results regarding strength were observed:

- *all-nodes* includes *all-nodes-r* and *all-nodes-s*;
- *all-nodes-s* includes *all-nodes-r* (and vice-versa);
- *all-edges* includes *all-nodes*, *all-nodes-r*, *all-nodes-s* and *all-edges-s*;
- *all-edges-s* includes *all-nodes-r* and *all-nodes-s*;
- *all-p-uses* includes *all-nodes*, *all-nodes-r* and *all-nodes-s*.

Based on these results, the null hypothesis $NH3$ can be rejected and the alternative hypothesis $AH3$ is accepted, indicating that the criteria can be complementary.

3.5. Discussion of Results

It is desirable that a test strategy presents a high effectiveness with a low cost and, therefore, these measures are strongly related. Thus, a testing criterion with high effectiveness can be prohibitive to apply if the cost is expensive. Leading in consideration the results obtained, we have proposed a test strategy to apply these testing criteria.

Figure 4 illustrates a partial order to apply the criteria. We have suggested that the tester starts the tests with the criteria *all-s-uses (a-s-u)*, *all-p-uses (a-p-u)* or *all-c-uses (a-c-u)*. From one of these criteria, the others can be

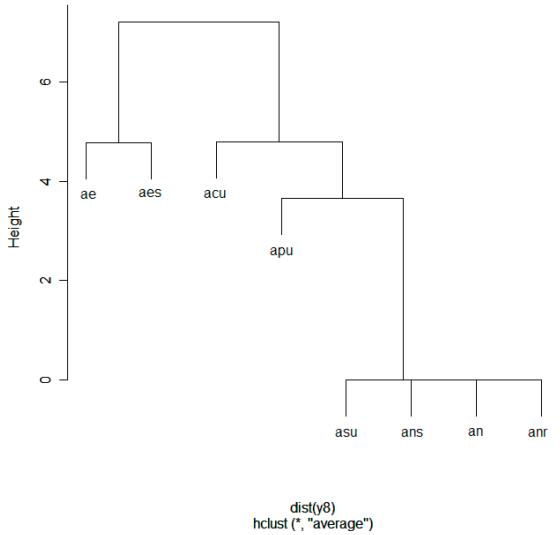

Fig. 3. Strength of the all-s-uses criterion.

added according to the flow suggested in the Figure 4. An important aspect is that each testing criterion considers different aspects for the tests, such as use of variables (*all-p-uses, all-c-uses*), communication between processes (*all-s-uses, all-edges-s, all-nodes-r, all-nodes-s*) or sequential control flow (*all-edges, all-nodes*). Using the order presented in Figure 4, the tester can choose the testing criteria according to which information he/she desires to cover in the concurrent program, improving the quality of the test activity.

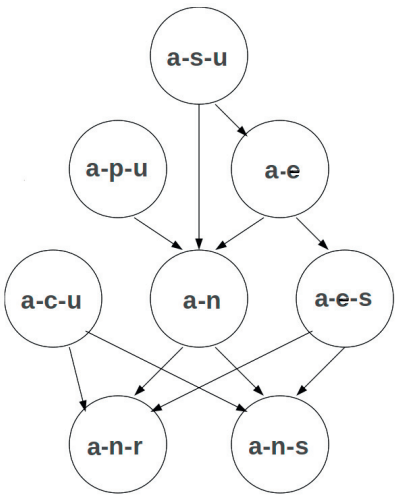

Fig. 4. A possible order to apply the testing criteria.

3.6. Threats to Validity

As for every empirical study, it is important to identify the threats to the validity of the reported results. Based on the principles presented by Wohlin [8] we have identified the following threats for our study:

Construction Validity: the ValiMPI testing tool automatically generates measurement about the testing criteria, avoiding the threat of satisfaction of some hypothesis or a particular criterion favored by the researcher.

Internal Validity: a factor that can compromise this validity is the bias that may occur during the insertion of defects in the programs. As the researcher knows the testing criteria, he could insert defects that would be easily revealed by the criteria. This threat was mitigated by the application of classifications of defects presented in the literature, in which the defects were inserted following a strategy similar to mutation testing strategy. Besides, the programs used in the study were not developed by the researcher involved.

Conclusion Validity: in this study the researcher might influence the results by trying to obtain the specific result. This threat was mitigated by the application of statistical methods to verify the normality of the data set, applying the methods according to the data set distribution.

4. Related Work

Recently, there has been significant progress in the understanding of the strength and limitations of concurrent programs testing. However, there have been few contributions concerned with experimental studies to evaluate testing criteria or compare testing tools in this context.

In [18, 19, 20] the authors evaluate the reachability testing and VeriSoft and RichTest tools and in [21] a technique for systematic exploration of thread interleavings is presented. The authors describe experiments to evaluate the technique.

In [22] a new technique for bugs finding in concurrent programs, called race-directed random testing (or RaceFuzzer) is proposed. The experimental results demonstrate that RaceFuzzer is able to create real situations of race conditions with very high probability to find them.

An experimental study to evaluate an abstraction-guided symbolic execution technique that detects errors in concurrent programs, is described in [23]. Five multi-threaded Java programs were used in the study. The results show that the technique generates feasible execution paths and finds concurrency errors quickly when compared to exhaustive symbolic execution testing.

In a similar study, we have investigated the same factors for other testing criteria, proposed for shared-memory programs [24]. Considering similar hypotheses, we have analysed the cost, effectiveness and strength of the testing criteria for multithreaded programs, implemented in (POSIX Threads). Differently, in this study we have compared sequential testing criteria and concurrent testing criteria in order to observe which group presents best effectiveness and minor cost. The results indicate that the concurrent testing criteria present a minor cost and are able to reveal some kind of defects, which are not revealed for sequential testing criteria.

In the context of MPI programs we have not identified studies that propose other structural testing criteria. In [25] the authors used formal methods that are very different from the experimental study presented here.

5. Conclusions and Future Works

This paper presented an experimental study to evaluate a family of structural testing criteria for message-passing programs. The objective was finding evidences about the cost, effectiveness and strength of those testing criteria, in order to define a testing strategy to apply them. The study was conducted considering the Experimental Software Engineering Process defined by Wholin [8].

The results show that the cost of the testing criteria based on control and communication flows is in general lower than the cost of testing criteria based on data flow and message-passing. These results are similar to those for sequential programs. In relation to effectiveness to reveal faults, the data analysis indicates that the criteria *all-s-uses* and *all-edges* are effective to reveal faults. Also, we have observed that some faults are only revealed for some criteria, indicating that criteria with minor effectiveness are relevant to reveal specific defects.

In relation to strength analysis, the *all-s-uses* is the strongest criterion, may include other criteria. The results of the strength indicate the complementary relationship among the testing criteria, therefore we have proposed an application test strategy.

As part of future work, we intend to compare our testing criteria with other testing approaches, for instance, model-check or mutation testing. Considering the recent work about mutation testing for MPI programs, presented

in [26], we intend to use the mutation testing to generate the faulty program versions in order to evaluate the effectiveness of our testing criteria.

According to Wohlin et al. [8], as an experiment will never provide a final answer to a question, it is important to facilitate its replication. In this sense, one of the contributions of this study is the packing of the material generated in the experiment. This lab package is available for public access, allowing a further evaluation, considering other testing mechanisms for message-passing programs.

Acknowledgements

The authors would like to acknowledge the Brazilian funding agency FAPESP for the financial support, under process 2009/04517-2.

References

[1] R. Taylor, D. Levine, C. Kelly, Structural testing of concurrent programs, IEEE Trans. on Software Engineering 18 (3) (1992) 206–215. doi:10.1109/32.126769.
[2] R.-D. Yang, C.-G. Chung, Path analysis testing of concurrent programs, Inf. Softw. Technol. 34 (1) (1992) 43–56.
[3] W. Wong, Y. Lei, Reachability graph-based test sequence generation for concurrent programs, International Journal of Software Engineering and Knowledge Engineering 18 (6) (2008) 803–822.
[4] R. Carver, Y. Lei, A general model for reachability testing of concurrent programs, International Conference on Formal Engineering Methods 3308 (2004) 76–98.
[5] Y. Lei, R. H. Carver, R. Kacker, D. Kung, A combinatorial testing strategy for concurrent programs, Software Testing Verification and Reliability 17 (4) (2007) 207–225.
[6] E. J. Weyuker, The cost of data flow testing: An empirical study, IEEE Trans. Softw. Eng. 16 (2) (1990) 121–128.
[7] W. E. Wong, A. P. Mathur, J. C. Maldonado, Mutation versus all-uses: An empirical evaluation of cost, strength and effectiveness, in: Software Quality and Productivity: Theory, practice and training, Chapman & Hall, Ltd., London, UK, UK, 1995, pp. 258–265.
[8] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén, Experimentation in software engineering: an introduction, Kluwer Academic Publishers, Norwell, MA, USA, 2000.
[9] M. P. I. Forum, Mpi: A message-passing interface standard - version 2.2, Tech. rep., University of Tennessee, Knoxville, Tennessee (2009).
[10] S. R. S. Souza, R. Vergilio, P. S. L. Souza, S. Simão, A. C. Hausen, Structural testing criteria for message-passing parallel programs, Concurrency Computation Practice and Experience. 20 (16) (2008) 1893–1916.
[11] A. C. Hausen, S. R. Vergilio, S. R. S. Souza, P. S. L. Souza, A. S. Simão, A tool for structural testing of MPI programs, in: 8th IEEE Latin-American Test Workshop, Cuzco, Lima, 2007.
[12] D. R. F. Bonetti, A. C. B. Delbem, G. Travieso, P. S. L. Souza, Optimizing van der waals calculi using cell-lists and MPI, in: IEEE Congress on Evolutionary Computation, IEEE, 2010, pp. 1–7.
[13] H. Krawczyk, B. Wiszniewski, Classification of software defects in parallel programs, Tech. rep., Faculty of Eletronics, Technical University of Gdansk, Poland (1994).
[14] M. J. Quinn, Parallel Programming in C with MPI and OpenMP, McGraw-Hill Education Group, 2003.
[15] A. Grama, G. Karpys, V. Kumar, A. Gupta, Introduction to parallel computing, 2nd Edition, Addison Wesley, 2003.
[16] J. DeSouza, B. Kuhn, B. R. Supinski, V. Samofalov, S. Zheltov, S. Bratanov, Automated, scalable debugging of MPI programs with intel message checker, in: International Workshop on Software Engineering for High Performance Computing System Applications, New York, NY, USA, 2005, pp. 78–82.
[17] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, E. H. Spafford, Design of mutant operators for the C programming language, Tech. rep., Software Engineering Research Center/Purdue University, sERC-TR-41-P (1989).
[18] C. R. Lei, Y., A new algorithm for reachability testing of concurrent programs, in: 16th IEEE International Symposium on Software Reliability Engineering, 2005, pp. 10–355.
[19] C. R. Lei, J., A stateful approach to testing monitors in multithreaded programs, in: 12th IEEE International Symposium on Higth-Assurance Systems Engineering, 2010, pp. 54–63.
[20] R. H. Carver, Y. Lei, Distributed reachability testing of concurrent programs, Concurr. Comput. : Pract. Exper. 22 (18) (2010) 2445–2466.
[21] P. Fonseca, C. Li, R. Rodrigues, Finding complex concurrency bugs in large multi-threaded applications, in: Conference on Computer Systems, EuroSys '11, New York, NY, USA, 2011, pp. 215–228.
[22] K. Sen, Race directed random testing of concurrent programs, SIGPLAN Not. 43 (6) (2008) 11–21.
[23] N. Rungta, E. G. Mercer, W. Visser, Efficient testing of concurrent programs with abstraction-guided symbolic execution, in: International SPIN Workshop on Model Checking Software, Berlin, Heidelberg, 2009, pp. 174–191.
[24] S. M. Melo, S. R. S. Souza, P. S. L. Souza, Structural testing for multithreaded programs: An experimental evaluation of the cost, strength and effectiveness, in: International Conference on Software Engineering and Knowledge Engineering, San Francisco, USA, 2012, pp. 476–479.
[25] S. S. Vakkalanka, S. Sharma, G. Gopalakrishnan, R. M. Kirby, ISP: a tool for model checking MPI programs, in: ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '08, New York, NY, USA, 2008, pp. 285–286.
[26] R. A. Silva, S. R. S. Souza, P. S. L. Souza, Mutation testing for concurrent programs in MPI, in: 13th Latin American Test Workshop, Quito, Ecuador, 2012, pp. 69–74.